

*Draw It or Lose it Web Application*

# **CS 230 Project Software Design Template**

Version 3.0

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_Toc115077317)

[**Table of Contents 2**](#_Toc115077318)

[**Document Revision History 2**](#_Toc115077319)

[**Executive Summary 3**](#_Toc115077320)

[**Requirements 3**](#_Toc115077321)

[**Design Constraints 3**](#_Toc115077322)

[**System Architecture View 4**](#_Toc115077323)

[**Domain Model 4**](#_Toc115077324)

[**Evaluation 5**](#_Toc115077325)

[**Recommendations 7**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 09/22/24 | Alex Leet | Initial creation of the software design document. Included the executive summary, detailed domain model description, and initial design constraints. |
| 2.0 | 10/06/24 | Alex Leet | Evaluation table that highlights the characteristics, benefits, and challenges associated with each platform (Mac, Linux, Windows, Mobile Devices) for the deployment of the gaming app. |
| 3.0 | 10/20/24 | Alex Leet | Added recommendations for platform, architecture, storage, memory, distributed systems, and security for *Draw It or Lose It* expansion. |

**Instructions**

Fill in all bracketed information on page one (the cover page), in the Document Revision History table, and below each header. Under each header, remove the bracketed prompt and write your own paragraph response covering the indicated information.

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room has requested the expansion of their Android-based game *Draw It or Lose It* into a web-based, multi-platform application to increase accessibility and engage a broader audience. The game features a team-based puzzle format where players guess images rendered in real-time. The primary goal of this project is to develop a scalable, high-performance solution that supports web-based access across multiple platforms while maintaining the core gameplay experience.

This document outlines the requirements for the project, evaluates different platforms for both hosting and client-side functionality, and describes the technical architecture needed to ensure the game operates smoothly across all devices. Additionally, it offers recommendations on the tools, technologies, and best practices to deliver a secure, reliable, and cost-effective solution for expanding *Draw It or Lose It*.

## Requirements

The following business and technical requirements outline the essential needs for expanding *Draw It or Lose It* into a web-based, multi-platform game.

* **Platform Accessibility**: The game must be expanded to support multiple platforms, including web browsers on both desktop and mobile devices.
* **Multi-Team Structure**: Each game can include one or more teams, and each team can have multiple players.
* **Unique Names**: Game, team, and player names must be unique to avoid conflicts in the system.
* **Single Game Instance**: There must be only one active instance of the game in memory at any given time to ensure consistency.
* **Real-Time Performance**: The game must maintain smooth, real-time interaction, especially during timed rounds of play.
* **Data Synchronization**: The system must ensure all players receive real-time updates so that game data is consistently synchronized across all devices.
* **Security Requirements**: The game must implement measures to protect user data and ensure secure communication across platforms.

## [Design Constraints](#_2et92p0)

When developing the game application in a web-based, distributed environment, several design constraints must be considered.

1. **Platform Compatibility**:

The game must function smoothly across various operating systems, including Linux, Mac, Windows, and mobile platforms like iOS and Android. This requires managing technical differences such as performance, security, and system resources to ensure consistent functionality across all devices.

1. **User Interface Flexibility**:

The user interface (UI) must adapt to different screen sizes and input methods, ensuring the game is easy to use on both desktops and mobile devices. Responsive design is needed to maintain a consistent and intuitive experience, regardless of whether players are using a mouse, keyboard, or touch screen.

1. **Real-Time Performance**:

The game must handle real-time gameplay, where multiple teams play simultaneously without notable delays. The server needs to manage large numbers of players and provide instant updates to ensure smooth gameplay across all platforms.

1. **Data Synchronization**:

To ensure a fair and consistent game experience, data must be synchronized in real-time for all players. Regardless of device or internet speed, everyone must see the same updates at the same time to avoid discrepancies during gameplay.

1. **Security**:

The game must protect user data and ensure secure communication between players and the server. This includes safeguarding personal information and preventing unauthorized access, while applying security measures consistently across all platforms.

1. **Licensing and Development Costs**:

Different platforms have varying costs for development and hosting. For example, Windows Server requires paid licenses, and iOS requires an Apple Developer Account for deployment. Balancing these costs with project efficiency is critical for staying within budget.

1. **Managing Game Instances**:

The system must ensure that only one version of the game is running at any given time. This prevents data conflicts and ensures a consistent experience for all players, even with multiple teams involved.

## [System Architecture View](#_ilbxbyevv6b6)

The game application will follow a three-tier architecture, which ensures that different parts of the system are separated for better organization and scalability.

1. **Client Tier**: This is where players interact with the game, using web browsers on desktops or mobile devices. The client handles the game’s user interface, sending actions (like guesses or team choices) to the server, and displaying real-time updates to the players.
2. **Server Tier**: The server is responsible for running the game’s logic. It manages things like player actions, team updates, and game rules. It also ensures that all players are seeing the same game state at the same time. The server handles requests from the clients and sends back game updates, making sure everything runs smoothly. It also ensures that only one version of the game is running.
3. **Data Tier**: The database stores all important game information, like player data, teams, and game history. The server talks to the database to get and update this information. This helps keep everything synchronized so all players see the same data.

Using this three-part structure makes the system easier to maintain, allows for better performance, and helps the game run smoothly for multiple users at once.

## [Domain Model](#_8h2ehzxfam4o)

The domain model shows how different parts of the game are structured and how they interact with each other. It’s designed using object-oriented principles to make the system more organized and efficient.

* **Entity Class**: This is the base class that holds common attributes like `id` and `name`. Other classes, like `Game`, `Team`, and `Player`, inherit from this class. This reduces the need to repeat code and helps keep things clean and simple.
* **GameService**: This class ensures that only one version of the game is running at any time. It uses the Singleton pattern, which controls the creation of the game instance. This helps maintain a single game in memory, preventing conflicts.
* **Game Class**: The `Game` class is responsible for managing teams. A game can have several teams, and these teams are stored in a list. This shows a one-to-many relationship because one game can have many teams.
* **Team Class**: Each team has multiple players, which are also stored in a list. Like the `Game` class, this is another example of a one-to-many relationship, as each team can have many players.
* **Player Class**: Each player is part of a team and has unique attributes like a name and ID.
* **ProgramDriver** and **SingletonTester**: The `ProgramDriver` class is responsible for starting the game, while `SingletonTester` checks that the Singleton pattern is working correctly, ensuring that only one instance of the game is created.

By using inheritance and relationships between these classes, the model keeps the design organized and easy to expand. This helps ensure the game can handle multiple teams and players efficiently while maintaining a single game instance.

**"The Gaming Room UML diagram. The top of the diagram is labeled as com dot gamingroom. Test boxes are placed in two layers. The first layer has three text boxes and the second layer has four of them. In the first layer, the 'ProgramDriver' textbox points to 'SingletonTester' textbox. The 'ProgramDriver' textbox contains the text 'asterisk main round brackets.' The 'SingletonTester' textbox contains the text 'asterisk testSingleton round brackets.' The arrow between these two text boxes are labeled 'open two angle brackets uses close two angle brackets'. In the second layer, there are 'GameService', 'Game', 'Team', and 'Player' text boxes. The 'GameService' textbox has texts arranged in two layers. The first layer contains games colon List open angle bracket Game close angle bracket, nextGamesId colon long, nextPlayer Id colon long, nextTeamId colon long, and service colon GameService. The second layer contains GameService round brackets, getinstance round brackets colon GameService, addGame open parenthesis name colon String close parenthesis colon Game, getGame open parenthesis id colon long close open parenthesis colon Game, getGame open open parenthesis name colon String close open parenthesis colon Game, getGameCount round brackets colon int, getNextPlayerID round brackets colon long, and getNextTeamId round brackets colon long. The 'GameService' box is connected with the 'Game' textbox with a line labeled 'zero dot dt dot asterisk'.  The 'Game' textbox also contains text in two layers. The first layers contains the text teams colon List open angle bracket Team close angle bracket. The second layer has Game open round bracket id colon long comma name colon String close parenthesis, addTeam open parenthesis name colon String close parenthesis Team, toString round brackets colon String. The 'Game' textbox is connected with the 'Team' textbox with a line labeled 'zero dot dt dot asterisk'. The 'Team' textbox also contains text in two layers. The first layers contains the text players colon List open angle bracket Player close angle bracket. The second layer has Team open parenthesis id colon long comma name colon String close parenthesis, addPlayer open parenthesis name colon String close parenthesis colon Player, and toString round brackets colon String. The 'Team' textbox is connected with the 'Player' textbox with a line labeled 'zero dot dt dot asterisk'. It contains the text Player open parenthesis id colon long comma name colon String close parenthesis and toString round brackets colon String. The 'Game', the 'Team, and the 'Player' boxes point to the 'Entity' textbox in first layer. The 'Entity' textbox contains text in two layers. The first layer has the text id colon long and name colon String. The second layer has Entity round brackets, Entity open parenthesis id colon long comma name colon String close parenthesis, getId round brackets colon long, getName round brackets colon String, toString round brackets colon String.**

## [Evaluation](#_2o15spng8stw)

Using your experience to evaluate the characteristics, advantages, and weaknesses of each operating platform (Linux, Mac, and Windows) as well as mobile devices, consider the requirements outlined below and articulate your findings for each. As you complete the table, keep in mind your client’s requirements and look at the situation holistically, as it all has to work together.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | | **Mobile Devices** |
| --- | --- | --- | --- | --- | --- |
| **Server-Side** | - **Characteristics**: Mac is better for smaller applications but not ideal for large web-based systems.  - **Advantages**: Works well with other Apple products and Services.  - **Weaknesses**: More expensive due to the need for Apple hardware and limited for larger businesses. | - **Characteristics**: Popular for running websites, works well for big companies.  - **Advantages**: Free, open-source, customizable.  - **Weaknesses**: Requires advanced technical expertise for setup and maintenance. | - **Characteristics**: Often used by businesses to run websites using Windows tool.  - **Advantages**: Great support for large organizations, easy to use with other Microsoft products.  - **Weaknesses**: Higher cost because you need licenses for the software. | | - **Characteristics**: Mobile devices don't host websites themselves but access them through web browsers.  - **Advantages**: Easily accessible through browsers on iOS and Android.  - **Weaknesses**: Needs strong optimization for mobile performance. |
| **Client Side** | - **Considerations**: Make sure the app works well with Safari, the main web browser for Mac.  - **Cost**: Higher due to the need for Apple-specific hardware and tools.  - **Time**: Moderate, with the need to ensure the application works well across different browsers. | - **Considerations**: Focus on making the app work smoothly on common Linus browsers like Firefox and Chrome.  - **Cost:** Low, as Linux and its tools are free to use.  - **Time:** Relatively minimal, as testing tools for Linux are widely available and easy to use. | - **Considerations**: Compatibility with browses like Edge and Chrome is essential, along with support for different versions of Windows.  - **Cost:** Higher due to licensing fees for Windows development tools.  - **Time:** Moderate to high, as ensuring compatibility across Windows versions can take additional time. | | - **Considerations**: Needs to be optimized for mobile performance, with a focus on touch interfaces and different screen sizes.  - **Cost:** Varies depending on whether a mobile-specific version or a web app is developed.  - **Time:** Higher, as testing across multiple mobile devices and platforms is necessary. |
| **Development Tools** | - **Languages/Tools**: Development on Mac uses web technologies like JavaScript, HTML, and CSS, with Xcode for iOS-related tasks.  - **Licensing Costs**: Higher, due to the requirement for Apple hardware and Xcode tools. | **- Languages/Tools:** Uses common development languages like Java, Python, and PHP, with free tools like VS Code and Eclipse.  - **Licensing Costs**: No significant costs, as most tools are free and open-source. | | **- Languages/Tools:** Windows development often involves .NET, C#, and Java, with Visual Studio as the main development environment.  - **Licensing Costs**: Higher, with costs associated with Visual Studio and Windows Server. | **- Languages/Tools:** Mobile development may involve JavaScript, React Native, and HTML5, with Android Studio for Android and Xcode for iOS.  - **Licensing Costs**: Free for Android, but higher for iOS due to the requirement for Apple hardware. |

## 

## Recommendations

The following recommendations address the critical technical decisions needed to expand *Draw It or Lose It*. Each suggestion supports the game's ability to scale while maintaining strong performance and security.

1. **Operating Platform**: I recommend Linux as the main operating platform for expanding *Draw It or Lose It*. Linux is widely used for hosting web applications because it’s reliable, scalable, and cost-effective. It’s also open source, which means there are no licensing fees, and it offers strong performance even as the game grows. Linux distributions like Ubuntu or CentOS are commonly used for web hosting, offering long-term support and strong community backing, which helps with troubleshooting or updates.

*Recommendation*: Use Linux as the main operating platform to ensure reliable performance, cost-effectiveness, and scalability as the game expands to more platforms and users.

1. **Operating Systems Architectures**: Linux supports a multi-tier architecture, where different parts of the system (like the user interface, game logic, and database) are separated. This structure makes it easier to scale, update, and manage each part of the system independently, ensuring that the game runs efficiently even as it expands to more players and platforms.

*Recommendation*: The multi-tier architecture of Linux is well-suited for *Draw It of Lose It*, as it allows for easier updates and scalability while maintaining smooth gameplay for users.

1. **Storage Management**: For storing game data (like player details, teams, and game scores), I recommend using a database system like MySQL or PostgreSQL. These databases are reliable, easy to manage, and work well with Linux. They can handle large amounts of data and keep everything organized, ensuring the game runs smoothly.

*Recommendation*: Implement MySQL or PostgreSQL to ensure reliable, scalable storage management with strong data integrity for handling player and game data.

1. **Memory Management**: Linux is known for its efficient memory management. It automatically allocates memory where needed, which helps the game run efficiently even when with many players active at the same time. Linux’s memory handling techniques, such as caching and virtual memory, ensure that the game can scale without performance issues.

*Recommendation*: Use Linux's memory management features to ensure efficient resource allocation and maintain smooth performance, especially during high traffic periods.

1. **Distributed Systems and Networks**: To make sure that *Draw It or Lose It* can run smoothly across different platforms, we’ll use distributed systems, with parts of the game running on different servers working together. When a player makes a move, the information is sent to the server and updates are shared with all players in real time, using technologies like WebSockets or RESTful APIs to keep everyone in sync. Load balancers can be used to handle traffic and avoid downtime by distributing the work across multiple servers. If one server goes down, another takes over, ensuring the game stays up and running.

*Recommendation*: Implement a distributed system using load balances and technologies like WebSockets or RESTful APIs to ensure real-time updates and smooth communication across platforms.

1. **Security**: I recommend using secure communication methods like HTTPS to encrypt data sent between players and the server. Additionally, strong authentication methods (like OAuth or JWT) will ensure that only authorized users can access game data. Linux offers strong built-in security features to protect the server and data from unauthorized access.

*Recommendation*: Use HTTPS for encrypted communication and implement OAuth or JWT for secure user authentication. Leverage Linux's built-in security features like SELinux or AppArmor to protect server integrity and prevent unauthorized access.